Resumen Examen de Septiembre programación Concurrente.

Tema1: Introducción a la programación Concurrente.

- Hay que evitar la espera activa (se denomina **espera activa** a una técnica donde un proceso repetidamente verifica una condición, tal como esperar una entrada de teclado o si el ingreso a una sección crítica está habilitado).
- Accion atómica: desarrollada sin interferencias de otras tareas (cada una de las instrucciones en ensamblador)
- Serializabilidad: ejecución concurrente equivalente a algún entrelazado de acciones atómicas.
- Propiedades Prog.Concurrente:
    - A menudo: (no siempre)
        - Complejidad no importante
        - Terminación no esencial.
        - Resultados finales no existentes
    - Relevante:
        - Correción de los resultados parciales.
        - Existencia de res. deseados
    - Tres tipos de propiedades:
        - Seguridad: Deben cumplirse **siempre**
        - Vivacidad: Deben cumplirse **en algún momento**
        - Prioridad.
            - Relacionadas con vivacidad
            - Diferentes prioridades pueden dar inanición
            - A veces esto es lo requerido
            - De manera explícita, unas tareas pueden tener más derecho a ejecutarse que otras.
        - Inanición: procesos conspiran para que otro no progrese. (vivacidad)

(Revisar los artículos del tema 1) Recordatorio

Tema2: Tareas en ADA.

Tareas en Ada:

- Anónimas (sólo una de cada tipo)

```
Task Tarea_A;

Task body tarea_A is

Begin

--codigo

End Tarea_A;
```

- Como tipos:
  - Sin paso de parámetros

```
Task type Tipo_A;
Task body tipo_A is
Begin
--codigo
End Tipo_A;
```

    - Declaración de variable crea tarea (apuntada por variable) -> Tarea_A : Tipo_A;
    - En cierto sentido variable ≡ tarea
    - Varias tareas con código idéntico->

      Varias_tareas: **array** (1..NA) **of** Tipo_A;

  - Con paso de parámetros

```
Task type Tipo_A (Parametro : Natural);
Task body tipo_A is
Begin -- codigo
Put(Parametro); --ejemplo
End Tipo_A;
```

- Independientemente: punteros a tareas (arranque dinámico)
  - Idea: Crear variable cuando sea necesario.

```
Type Tipo_Punt_A is Access Tipo_A;

Una_Tarea : Tipo_punt_A;

Begin

Una_Tarea := new Tipo_A(3);

End;
```

- Excepciones:
    - Las excepciones en ADA no se propagan fuera de las tareas
    - Resultados inesperados
    - Programa no termina
    - Capturar excepciones en las tareas.

Tema3: Secciones críticas y exclusión mutua.

Conceptos previos:

Los algoritmos de **exclusión mutua** (mutex de mutual exclusion) se usan en programación concurrente para evitar que fragmentos de código conocidos como **secciones críticas** (solo una tarea a la vez) sean accedidos al mismo tiempo a recursos que no deben ser compartidos.

Sección crítica: Soluciones:

- Hardware: Instrucciones test-and-set y swap
- Software: algoritmos exclusión mutua y soporte del S.O

Ejemplos de intentos de exclusión mutua:

<table>
<tr><th colspan="2">Primer Ejemplo.</th></tr>
<tr><td colspan="2">type Tipo_Turno is (Derecha, Izquierda);
Turno : Tipo_Turno := Derecha;</td></tr>
<tr><td>loop
while (Turno /= Derecha) loop
null;
end loop;
….
Turno := Izquierda;
end loop;</td><td>loop
while (Turno /= Izquierda) loop
null;
end loop;
…
Turno := Derecha;
end loop;</td></tr>
<tr><td colspan="2">Propiedades:
Se consigue exclusion mutua.
No hay interbloqueos (deadlock)
No hay inanición (starvation)
Problemas de diseño:
- Cadencia de ejecución.
- Terminación(si la montaña de uno de los trenes tiene mas perímetro que la del otro y la velocidad de ambos es igual, entonces dicho tren tarda mas en volver)</td></tr>
</table>

Segundo Ejemplo.	
Entra_dcho : Boolean := False; Entra_izq : Boolean := False;	
loop while Entra_Dcho loop null; end loop; Entra_Izq := True; …. Entra_Izq := False; end loop;	Loop while Entra_izq loop null; end loop; Entra_Dcho := True; … Entra_Dcho := False; end loop;
Propiedades: No hay exclusión mutua (se puede dar el caso en el que ambos procesos pasen el while (entra_izq y entra_dcho ambos = true) y entonces no se cumpla la propiedad de exclusión mutua)	

Tercer Ejemplo.	
Entra_dcho : Boolean := False; Entra_izq : Boolean := False;	
Loop Entra_Izq := True; while Entra_Dcho loop null; end loop; …. Entra_Izq := False; end loop;	Loop Entra_Dcho := True; while Entra_izq loop null; end loop; … Entra_Dcho := False; end loop;
Propiedades: Interbloqueo (deadlock) Ejemplo: si entra el proceso P1, y pone Entra_Izq a True, y en ese momento entra el proceso P2 poniendo Entra_dcho := True, se quedaran ambos en bucle infinito.	

Cuarto Ejemplo.	
Entra_dcho : Boolean := False; Entra_izq : Boolean := False;	
Loop Entra_Izq := True; while Entra_Dcho loop Entra_Izq := False; Entra_Izq := True; end loop; …. Entra_Izq := False; end loop;	Loop Entra_Dcho := True; while Entra_izq loop Entra_Dcho := True; Entra_Dcho := False; end loop; … Entra_Dcho := False; end loop;
Propiedades: Soluciona el problema del interbloqueo Produce inanición.(starvation)	

<table>
<tr><th colspan="2">Quinto Ejemplo. (Algoritmo de Peterson)</th></tr>
<tr><td colspan="2">type Tipo_Turno is (Derecha, Izquierda);
Turno : Tipo_Turno := Derecha;
Entra_Izq := False;
Entra_dcha := False;</td></tr>
<tr><td>Loop
Entra_izq := True;
Turno := derecha;
while Entra_Der and (Turno = Derecha)
loop
null;
end loop;
…. Seccion critica
Entra_Izq := False;
end loop;</td><td>Loop
Entra_Dcho := True;
Turno := Izquierda
while Entra_izq and (Turno = Izquierda)
loop
null;
end loop;
… sección critica
Entra_Dcho := False;
end loop;</td></tr>
<tr><td colspan="2">Propiedades:
- Se consigue exclusión mutua (ya que Turno resuelve el problema de que Entra_der y Entra_Izq ambos = true (segundo ejemplo), ya que uno de ellos es el ultimo que pone turno a izq o a derecha)
- Cadencia correcta: Uno de los procesos no esta haciendo nada (Entra_proceso = false), entonces el otro proceso no esta bloqueado y puede continuar (ver Proceso Izq, si Enta_Der = false entonces sale del bucle y hace la sección critica)
- No hay interbloqueos, pues aunque se diera el caso del ejemplo 3, el turno o esta en derecha o en izquierda, por lo que nunca se queda en el bucle.
- Ausencia de inanición, Uno de los procesos entra en la sección critica repetidamente (al salir le da la oportunidad al otro proceso de entrar(con entra_elOtro = true)), podría darse el caso de que el primer proceso intentará reentrar antes que el otro, pero al intentar reentrar se quita el turno así mismo y además el otro ha solicitado entrar con (entra_elotro = true)</td></tr>
</table>

Espera Activa

- El algoritmo de Peterson y otros realizan espera activa.
- Tareas esperando entran a realizar trabajo.
- Validos como idea inicial (solo algunas veces, usados en casos muy particulares (multiprocesadores, R.C muy pequeñas, sistemas con hardware dedicado)
- No utiles en la mayor parte de los casos
- Mecanismos de mas bajo nivel (S.O suspende las tareas)
- ***Huid de la espera activa***

Resolver este ejercicio:

c1 , c2 : Integer := 1;
loop −− Proceso P1
loop
c1 := 1 − c2 ;
exit when c2 /= 0;
end loop ;
. . . . – Seccion critica
c1 := 1;
end loop ;
loop −− Proceso P2
loop
c2 := 1 − c1 ;
exit when c1 /= 0;
end loop ;
. . . . – Seccion criticaa
c2 := 1;
end loop ;
	Hay **exclusión mutua**, ya que nunca ambos procesos ejecutan la sección critica a la vez (o bien c1 := 1 o c2 := 1 condición de salida del bucle).
	
	No hay **interbloqueos** (la única posibilidad es que tanto c1 como c2 fueran 0 en algún momento, como cuando uno se hace 0, la otra condición no se puede hacer 0, llegamos a una contradicción, por lo que no hay interbloqueos).
	
	Ausencia de inanición, cuando uno de los procesos ejecuta la sección critica podría darse el caso de que intentará volver a entrar antes que el otro, pero al ejecutar la S.C ejecuta (c1 o c2 := 1) quitándose turno así mismo.
	
	Cadencia correcta, si uno de los procesos no hace nada, el otro de los procesos puede continuar ya que la condición de salida del bucle se cumple, por lo que no está bloqueado.

Tema4: Semáforos.

- Según Dijkstra es un Tipo abstracto de datos

Init (sem, n) ≡ (sem := n) Wait (sem) ≡ (**AWAIT** sem>0 --> sem := sem - 1) Signal (sem) ≡ (sem := sem + 1)

- Código entre ( ) -> acción atómica.
- Init (sem, n) ≡ solo inicializa los valores
- **AWAIT** *cond* -> suspende tarea hasta cumplimiento de cond
- Wait (sem) suspende hasta que sem>0 entonces decrementa
- Signal (sem) incrementa sem.

Exclusión mutua con semáforos.

X := …; -- compartida Init(Mutex,1); --nadie en S.C Wait(Mutex); X:=X+X: --S.C Signal(Mutex);

Si mutex = 0 entonces alguien en S.C

Wait

- puede decrementar si sección crítica está libre.
- Suspende en otro caso

Signal

- Siempre puede incrementar
- Puede rearrancar alguna tarea suspendida

Clases y e implementación.

- Binarios (sem € {0,1}) exclusión mutua
- Generales(sem € N) sincronización más avanzada.
- Semaforos binarios: Signal cuando ya tiene un valor 1 -> efecto indefinido

- EN ADA:
  - Inicializados al máximo valor por defecto
  - **Importante**: la tarea madre o una de las tareas hijas debe inicializarlo con el valor deseado si este no coincide con el máximo. (Init (sem, N))
  - **Mutex : Semaphores.Bin_Semaphore – semáforo binario inicializado a 1**

Ejemplos de uso de Semáforos.

<table>
<tr><th colspan="2">Primer Ejemplo.</th></tr>
<tr><td colspan="2">Mutex : Bin_Semaphore;
N_C : Tipo_Tam := 0;</td></tr>
<tr><td>-- tarea de salida
Loop
Barrera_Salida;
Wait (Mutex);
N_C := N-C -1 ;
Signal (Mutex);
Abrir_Salida;
End loop;</td><td>--Tarea entrada
loop
Barrera_entrada;
Espacio := False;
While not Espacio loop
Wait(Mutex)
If N_C < Max then
N_C := N_C + 1;
Espacio := True;
End if;
Signal (Mutex);
End loop;
Abrir_Entrada;
End loop;</td></tr>
<tr><td colspan="2">Propiedades.
Recurso compartido (N_C-> número de coches aparcados)
Acceso en exclusión mutua.
Si el aparcamiento está lleno (es decir, no se entra en el if N_C < max) se produce espera activa</td></tr>
</table>

<table>
<tr><th colspan="2">Segundo Ejemplo.</th></tr>
<tr><td colspan="2">Mutex, No_Lleno : Bin_Semaphore;
N_C : Tipo_Tam := 0;</td></tr>
<tr><td>-- tarea de salida
Loop
Barrera_Salida;
Wait (Mutex);
N_C := N_C -1 ;
If N_C = MAX – 1 then
Signal (No_Lleno);
End if;
Signal (Mutex);
Abrir_Salida;
End loop;</td><td>--Tarea entrada
loop
Barrera_entrada;
Wait(No_Lleno);
Wait(Mutex);
N_C := N_C + 1;
If N_C < MAX then
Signal(No_Lleno)
End if;
Signal (Mutex);
Abrir_Entrada;
End loop;</td></tr>
</table>

Propiedades.
Mutex protege la S.C No_lleno sincroniza (sincronización condicional) No_lleno = 0 -> N_C = MAX (se bloquea si el aparcamiento está lleno) No_Lleno = 0 sii ha entrado el ultimo coche No hace **espera activa**

Tercer Ejemplo.	
Huecos : Semaphore (max);	
-- tarea de salida **Loop** Barrera_Salida; Signal (Huecos) Abrir_Salida;	--Tarea entrada **loop** Barrera_entrada; Wait (Huecos); Abrir_Entrada; **end loop;**
Propiedades. Wait (huecos) permite paso si hay espacio Signal(Huecos) libera espacio Solo bloquea cuando sea 0 (es decir cuando no haya huecos)	

Tema5: Especificación formal de recursos compartidos.

- Recursos compartidos:
  - Tipo abstracto con:
    - Estado interno
    - Operaciones
    - Condiciones de sincronización.
    - Exclusión mutúa implicita
  - Notación alto nivel
  - Independente de la implementación.
- Razones para especificar:
  - Formalismo: un buen lenguaje es formal (no ambiguo) transmitiendo ideas de forma inequívoca y además es preciso. Usaremos lógica de primer orden
  - Independencia del lenguaje de programación:
  - Claridad y brevedad:
  - Demostrabilidad: Permite demostrar que se cumplen ciertas propiedades
- Especificacion de recursos. (esquema minimo)

C-TADSOL Nombre recurso

OPERACIONES

ACCION Operación_recurso: Tipo_recurso[es] x Tipo1[e] x .. TipoN [s]

SEMÁNTICA

DOMINIO

TIPO: Tipo_recurso = …

DONDE: Tipo_Adicional = …

INVARIANTE: ...

INICIAL (r): Formula que especifica el valor inicial del recurso

CPRE: Precondicion de concurrencia

CPRE: P (r, a1,…,an)

Operación_Recurso1 (r, a1,…,an)

POST: Postcondicion de la operación

POST: Q(r,a1,…an)

- Similitudes y diferencias entre recursos y TADS
    - Semejanzas:
        - Encapsulamiento de datos
        - Reusabilidad
        - Razonamiento corrección recurso independiente de su uso
        - Razonamiento uso correcto sin su implementación.
    - Diferencias
        - Estado recurso dependiente del mundo externo (tareas accediendo)
        - No noción igualdad, no copia
        - Notación aumentada
        - Algunos argumentos privilegiados: recurso único-> cambios propagados inmediatamente
- Declaración de interfaz
    - Nombre de operaciones + tipos y modos de argumentos
    - Modos: no necesarios pero ayudan a documentar ([e],[es],[s])
    - Tipos:
        - Básicos: B, N, Z, R
        - Algebraicos:
            - Tuplas: -> (Ta x Tb)

        - Constructores: -> nombre (Ta x Tb)
        - Union de tipos: -> (Ta | Tb)
    - Especiales:
        - Secuencias: -> Secuencia (Ta)
        - Conjuntos: -> Conjunto(Ta)
        - Tablas
        - Maps
        - Funciones parciales: -> Ta ->-> Tb
- Dominio : Tipo + Invariante
    - Tipo:Admitimos campos con nombres en tuplas
    - Invariante:
        - Magnitud o ley que se mantiene tras una transformación.
        - Verdad antes de la entrada y de la salida de cada operación
        - Puede violarse dentro de una operación.
        - Permite establecer propiedades generales (y trabajar sobre ellas)
- Estado Inicial: **INICIAL** (R) = In(R))
    - Determina valor inicial del recurso
    - Si fuera una operación separada:
        - Sería algo especial
        - Sólo se la llamaría una vez
        - Muchos lenguajes permiten especificar estado inicial
- Especificación de operaciones
    - **PRE:** Condiciones no relacionadas con la concurrencia
        - Determinan corrección llamada
        - Pueden causar excepciones, error de ejecución etc
    - **CPRE:** sincronización entre operaciones
        - Determinan sincronización
        - Pueden causar suspensión
        - Recurso cambiado por otras tareas
    - **POST:** postcondiciones (cambio en el recurso/argumentos)

Tema6: Memoria compartida: Objetos protegidos.

Esquema de un objeto Protegido:

Protected type Tipo **is** **Function** F1 (…); **Procedure** P1 (…); **Entry** E1 (…);	Parte visible del objeto protegido --declaraciones similares al ads (pero los tipos deben de estar ya definidos) Function: permite acceso concurrente. (solo de lectura, excluye a cualquier escritor) Procedure y entries: acceso en exclusión mutua. (tanto de lectura como escritura) No usaremos funciones y procedure es una clase de entry (con guarda when true)
Private Var_Estado : Tipo_Var; **Entry** E_Priv (…); **End** tipo;	No hay declaración de tipos Variables de estado (no visibles desde ext) Entry privadas: - Aparecen en la implementación - No se pueden llamar desde afuera
Protected body tipo **is** End tipo;	Contiene el código para los funtion, procedures, entries declaradas No declaraciones de tipos ni de **variables locales**, entry además recoge condiciones de sincronización. (when)

Sincronización condicional:

Entry Op (…) **when** condición **is**
begin

…

End Op;

- When condicion: guarda de la operación.
- Condicion: cualquier variable visible **excepto** argumentos de llamada
- Usar solo variables de estado
- Llamada a op (..) suspende si no se cumple la condición.

Comportamiento en suspensión.

- Si no se cumple condición: llamada se suspende (se abandona exclusión mutua)
- Otra llamada puede entrar
- Si el estado cambia, puede cumplirse condición
- Re arrancar alguna llamada suspendida (Ada no define cuál)
- El orden de evaluación de las guardas no esta definido

- Condición puede reevaluarse muchas veces
- Llamadas bloqueadas con guardas abiertas tienen preferencia sobre aceptación de nuevas llamadas
- Evaluación argumentos de Op (..) espera aceptación
- Conserva cantidad observable de trabajo (no hay computación especulativa)

Dependencia de parámetros de entrada

- cuando condición de sincronización (when ..) depende de un valor que no es parte del estado del recurso.
- Ada no ofrece mecanismos para esto pero nos ofrece:
    - Requeue
    - Familia de entries.
- Se podría decir que almacenan información sobre los parámetros de entrada en el estado privado del objeto.

REQUEUE

- Derivación a una entry privada.
- Diferente a llamar a una operación publica desde interior del objeto. (causaría autobloqueo)
- Ejemplo:
    - Entry P (..) when C is begin… requeue P1; …
    - Entry P_1 (..) when C_1 is begin …
        - Puede bloquear en P_1 (implica que abandona exclusión mutua)
        - Puede rearrancar más tarde (otra operación puede hacer verdad C_1)
        - Al acabar P_1:
            - **No** retorna a P;
            - Sale del objeto protegido
        - When C es opcional si luego C_1 es when C + resto de condiciones

FAMILIA DE ENTRIES

```
Type Tipo_X is …;

Protected type Tipo_P is

Entry P (Tipo_X);

…

End Tipo_P;

Protected body Tipo_P is

Entry P (For I IN tipo_X)

When Estado > I is begin

Q(I, …)

… End Tipo_P;
```

- **Tipo_X:** ha de ser escalar (natural, enumerado..)
- Sincronización con estado e índice
- Replica condición para cada I (como si hubiese un fragmento de código para cada I € Tipo_X)
- Las familias de entries admiten parámetros (no pueden formar parte de la guarda)
- Se evalúan las condiciones tras cambio de estado
- Reevaluación de llamadas suspendidas
- El tiempo aumenta con tamaño del rango

Tema7: Gestores de sincronización.

No importante, leer apuntes, recordar:

- Para saber el número de tareas bloqueadas en una entry (entry'count)

Tema8: Paso de mensajes: Rendez-Vous

- Motivación:
    - Sistemas distribuidos
        - Ausencia de memoria compartida
        - Datos repartidos entre procesos / tareas
        - Quizás en maquinas distintas
        - Se accede a datos mediante comunicación de procesos : paso de mensajes
        - También sirve para sincronización
    - Aparecen al conectar ordenadores a una red
    - No hay exclusión mutua
    - Problema: comunicación entre procesos
- Filosofía cliente-servidor
    - Proceso servidor se ejecuta en un ordenador
    - Proceso cliente se ejecuta donde sea más conveniente al usuario
    - Procesos resultantes del análisis darán lugar a procesos cliente
- Canales de comunicación
    - Los procesos se ejecutan en espacios de memoria independientes y se comunican intercambiando mensajes, que contienen copias de variables del emisor
        - Send (canal, mensaje)
        - Receive (canal, mensaje)
- Clasificación:
    - Por el comportamiento dinámico:
        - Síncrono
        - Asíncrono
    - Nombrado de los canales
        - **Explícito**: los canales son un tipo de datos del lenguaje y el programador debe declararlos, asignarlos, destruirlos, etc
            - Ventajas:
                - Pueden declararse vectores o matrices de canales
                - Pasar canales como datos: útil en entornos de cliente-servidor
            - Inconvenientes:
                - Mas trabajo para el programador

        - **Implícito**: Canales ocultos al programador: se identifican proporcionando un nombre del proceso receptor.
            - Ventajas:
                - Menos código, no es necesario declarar ni destruir
            - Inconvenientes:
                - Sólo permiten comunicación 1:1
    - Por la cardinalidad de los esquemas de comunicación
        - **1:1** cada canal tiene un emisor y un receptor
        - **N:1** varios procesos pueden enviar mensajes por el canal y solo uno puede recibirlos, el habitual admite nombrado explícito (puertos)
        - **N:M** varios emisores y varios receptores (no mucho éxito)

Sincronicidad de los canales

- ***La recepción siempre es bloqueante***
- **Asíncronos**: ***Envío no bloqueante***, el emisor sigue ejecutando sin esperar a que la comunicación tenga efecto.
    - Ventajas: mayor concurrencia: útil si tiempo de respuesta es elevado (telegramas, emails)
    - Inconvenientes: Necesita colas de mensajes (puede ser costoso)
- **Síncronos**: ***Envío bloqueante***, emisor y receptor se sincronizan para intercambiarse cada mensaje (hablar por teléfono, en persona …)
- Mayoría de lenguajes *optan por canales síncronos*,
- Canales asíncronos: protocolo de comunicaciones (TCP/IP) o computación paralela

RENDEZ-VOUS EN ADA

El recurso compartido será propiedad de un proceso servidor, en el cuál declararemos una serie de recursos públicos para los procesos cliente.

```
Task type Tipo_Servidor is

Entry Operación1 (parámetros);

…

Entry OperacionN (parámetros);

End Tipo_Servidor;
```

El código del proceso servidor suele consistir en un bucle dentro del cual se ejecuta una estructura de recepción alternativa de peticiones (select)

```
Select

When Condicion =>

Accept Operacion1 (parámetros) do

… -- rendez vous

End;

--<sentencias fuera del rendez vous>

Or

When condición =>

Accept Operación…..

….

Or

When condicion =>

Accept OperacionN (parametros) do

… -- rendez vous

End;

--<sentencias fuera del rendezvous>

End select;
```

FUNCIONAMIENTO DE LA SELECT:

En primer lugar se evalúan las guardas, y aquellas que resulten ser falsas son descartadas. Las clausulas accept cuya guarda ha evaluado a cierto son los servicios disponibles, en este instante, paras los clientes:

- Si en el momento de llegar al servidor a la ejecución de la select, algún cliente ha ejecutado una llamada a alguno de los servicios disponibles, se seleccionara uno de ellos.
- Si, por el contrario, ningún cliente ha comunicado todavía su interés en ninguno de los servicios disponibles, el servidor se quedará bloqueado y atenderá a la primera petición que reciba.

Cuando se selecciona uno de los servicios, se ejecuta exclusivamente el código correspondiente a su cláusula accept. Como se puede ver, este código consta de dos partes bien diferenciadas: el

- El rendez-vous (entre el do hasta el end que lo cierra)
- El resto del código para esa clausula

El rendez-vous es la zona de sincronización entre cliente y servidor: el cliente cuya petición está siendo atendida queda bloqueado hasta que el servidor alcanza el final del rendez-vous. Las instrucciones restantes ya pueden ser ejecutadas de manera concurrente por cliente y servidor, a diferencia de lo que ocurría con objetos protegidos

IMPORTANTE: Es importante tener esto en cuenta pues si se coloca alguna operación lenta, bloqueante o propensa a fallar en el rendez-vous se provocaría un problema de falta de concurrencia o interbloqueo al cliente. Como regla general este tipo de operaciones deben colocarse siempre fuera del rendez-vous.

Solo se pueden modificar los parámetros formales de la clausula accept dentro del rendez-vous. Las guardas solo pueden hacer referencia a variables de estado del servidor, nunca a parámetros formales del accept (parámetros de entrada).

Esquemas de código:

La situación más sencilla que nos podemos encontrar es cuando ninguna de las operaciones depende de los parámetros de entrada. En este caso el servidor del recurso tendrá una entry por cada operación del CTAD y un bucle principal donde se pondrán a disposición de los clientes aquellas operaciones cuya CPRE sea cierta:

```
Task type Tipo_servidor is

Entry Operacion1 (parámetros);

…

Entry OperacionN (parámetros);

End Tipo_Servidor;


Task body Tipo_Servidor is

-- declaración/ inicialización del estado del servidor

Begin

-- resto de la inicialización

loop

Select

When Condicion =>

Accept Operacion1 (parámetros) do

… -- rendez vous

End;

--<sentencias fuera del rendez vous>

Or

When condición =>

Accept Operación…..

….

Or

When condicion =>

Accept OperacionN (parametros) do

… -- rendez vous

End;

--<sentencias fuera del rendezvous>
```

```
End select;

End loop;

End Tipo_Servidor;
```

Si la CPRE depende de parámetros de entrada pasamos a **un esquema con bloqueo en dos fases** relativamente similar al que se usaba con objetos protegidos: una rama de la select con la guarda a cierto en la que se envía la información necesaria para evaluar la CPRE y un posterior bloqueo al cliente hasta que la CPRE se cumple y se decide servir la operación.

Para llevar a cabo este bloqueo en dos fases haremos uso del paquete genérico channels que proporciona un tipo de canales sencillos con operaciones Send y receive (nombrado de canales explicito)

```
Task type Tipo_servidor is

Entry Operacion1 (parámetros);

…

Entry OperacionN (parámetros);

End Tipo_Servidor;


Task body Tipo_Servidor is

-- declaración/ inicialización del estado del servidor

Begin

-- resto de la inicialización

Loop

Select

…..

When True =>

Accept OperacionX (parámetros entrada + canals respuesta) do

--Almacenar petición
```

```
End;
Or
…
End select;
While hay peticiones pendientes que se puedan atender loop
Extraer (canalRespuesta, DatosPeticion)
realizarOperacion (if cpre lala lo que sea)
Send (CanalRespuesta, respuesta/confirmación),
End loop;
End loop;
End Tipo_Servidor;
```

El proceso cliente (consumidor) tendrá que ejecutar una llamada a la entry seguida de una recepción incondicional:

```
CResp : InstanciaDeChannel.Channel;
…
Tipo_Servidor.OperacionX (…, CResp);
Receive (CResp, Respuesta/confirmación);
```

El mensaje del servidor suele contener las variables de salida de la operación del recurso, en caso de existir, o una simple confirmación en caso contrario.

Aunque este es un esquema general de codificación del servidor, comúnmente se opta por esquemas mixtos, en los que las operaciones cuya CPRE no depende de los datos de entrada se sincronizan por la guarda y el resto usan canales explícitos para sincronizar al cliente.

Políticas explicitas: decidir cuál es la mejor petición a atender (modificar el bucle while del servidor) para saber cuál es la mejor